

**CSC 311 – Winter 2022-2023**  
**Design and Analysis of Algorithms**  
**1. Introduction and Mathematics**  
**essentials**

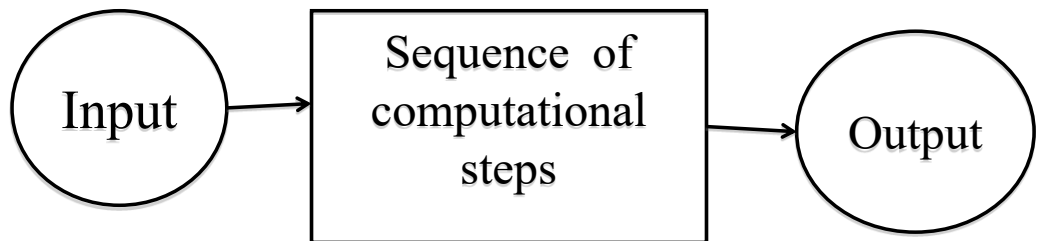
Prof. Mohamed Menai  
Department of Computer Science  
King Saud University

# Outline

- Algorithms
- Correct algorithms
- Data structures
- Technique
- Hard problems
- Choosing algorithms
- Design of algorithms
- Analysis of algorithms
- Mathematical essentials

# Algorithms

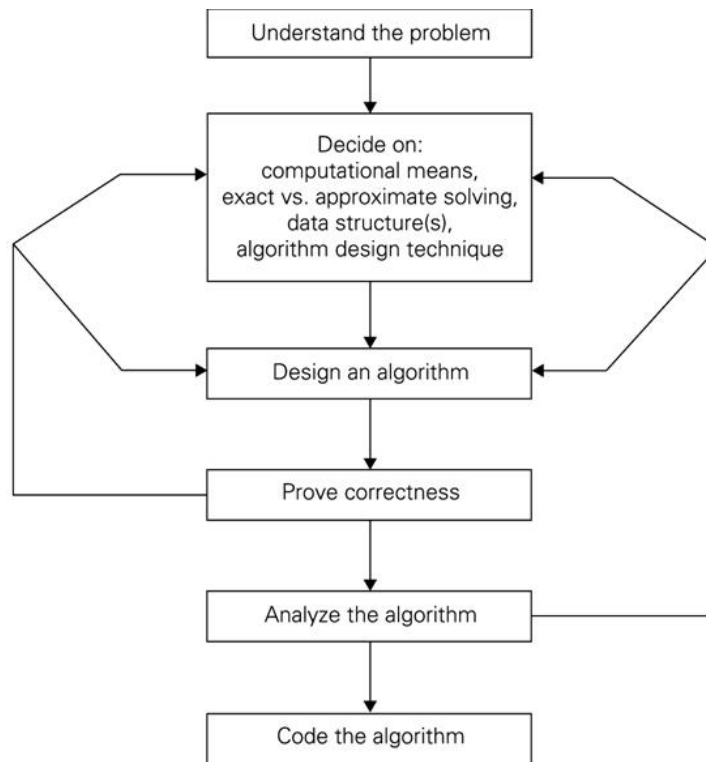
An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.



## Example: Sorting problem

- Input: A sequence of  $n$  numbers:  $a_1, a_2, \dots, a_n$
- Output: A permutation (reordering)  $a_i, a_j, \dots, a_k$  of the input sequence such that  $a_i \leq a_j \leq \dots \leq a_k$
- Ex. Input: sequence 31, 41, 59, 26, 41, 58  
Output: sequence 26, 31, 41, 41, 58, 59

# Design and Analysis of an Algorithm



## Correct Algorithms

- An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem.
- An incorrect algorithm might not halt at all on some input instances, or it might halt with an answer other than the desired one.

## What kinds of problems are solved by algorithms?

- We are given a road map on which the distance between each pair of adjacent intersections is marked, and our goal is to determine the shortest route from one intersection to another.
- We are given a sequence  $A_1, A_2, \dots, A_n$  of  $n$  matrices, and we wish to determine their product  $A_1 \cdot A_2 \cdot \dots \cdot A_n$
- We are given an equation  $ax \equiv b \pmod{n}$ , where  $a$ ,  $b$ , and  $n$  are integers, and we wish to find all the integers  $x$ , modulo  $n$ , that satisfy the equation.
- We are given  $n$  points in the plane, and we wish to find the convex hull of these points (the smallest convex polygon containing the points).

## Data structures

- A data structure is a way to store and organize data in order to facilitate access and modifications.
- No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them:
  - Table, Stacks and Queues, Linked lists
  - Representing rooted trees
  - Hash tables
  - Binary Search Trees
  - Red-black trees, ...



## Technique

- Techniques of algorithm design and analysis are used to develop algorithms, to show that they give the correct answer, and to understand their efficiency.

## Hard problems

- There are some problems for which no efficient solution is known, which are known as NP-complete:
  - It is unknown whether or not efficient algorithms exist for NP-complete problems.

## Pseudocode conventions

- **Algorithm** indicates the beginning of the algorithm.
- Indentation indicates block structure.
- **while**, **for**, **repeat-until**, and **if-else** have same interpretations similar to Java.
  - for  $i=1$  to  $n$  {  
     $x = x+i$   
     $y = y+x$   
    ...}
- `//` indicates the beginning of a comment
- Array  $A[1..n]$ ; array elements:  $A[i]$ ,  $A[i+2]$  ...
- **return** statement returns the control back to the point of call in the calling procedure.

## Choosing algorithms

Ex: Fibonacci sequence is defined as follows.

$$F(0) = 0, F(1) = 1, \text{ and}$$

$$F(n) = F(n-1) + F(n-2) \text{ for } n > 1.$$

Write an algorithm to compute  $F(n)$ .

There are many algorithms, but what is the most efficient one?

## Algorithms 1 and 2 for Fibonacci

```
function fib1(n){  
    if n < 2 then return n;  
    else return fib1(n-1) + fib1(n-2);  
}
```

```
function fib2(n){  
    i = 1; j = 0;  
    for k = 1 to n do { j = i+j; i = j- i;}  
    return j;  
}
```

## Algorithm 3 for Fibonacci

```
function fib3(n){  
  i = 1; j = 0; k = 0; h = 1;  
  while n>0 do {  
    if (n odd) then { t = jh;  
                     j = ih + jk +t;  
                     i = ik +t;}  
    t = h^2;  
    h = 2kh+t;  
    k = k^2+t;  
    n = n div 2;}  
  return j;  
}
```

## Example of running times for Fibonacci

n	10	20	30	50	100	1000 0	1 000 000	1000 0000 0
fib1	8 ms	1 s	2 min	21 days				
fib2	1/6 ms	1/3 ms	1/2 ms	3/4 ms	3/2 ms	150 ms	15 s	25 min
fib3	1/3 ms	2/5 ms	1/2 ms	1/2 ms	1/2 ms	1 ms	3/2 ms	2 ms

## An Example: Insertion Sort

```
InsertionSort(A, n) {  
  for i = 2 to n {  
    key = A[i]  
    j = i - 1;  
    while (j > 0) and (A[j] > key) {  
      A[j+1] = A[j]  
      j = j - 1  
    }  
    A[j+1] = key  
  }  
}
```



## An Example: Insertion Sort

30	10	40	20
1	2	3	4

$i = \emptyset$ $j = \emptyset$ $\text{key} = \emptyset$ $A[j] = \emptyset$ $A[j+1] = \emptyset$
---



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}
  
```

## An Example: Insertion Sort

30	10	40	20
1	2	3	4

$i = 2$	$j = 1$	$\text{key} = 10$
$A[j] = 30$	$A[j+1] = 10$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2$	$j = 1$	$key = 10$
$A[j] = 30$	$A[j+1] = 30$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2$	$j = 1$	$\text{key} = 10$
$A[j] = 30$	$A[j+1] = 30$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2$	$j = 0$	$\text{key} = 10$
$A[j] = \emptyset$	$A[j+1] = 30$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2$	$j = 0$	$\text{key} = 10$
$A[j] = \emptyset$	$A[j+1] = 30$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 2$	$j = 0$	$\text{key} = 10$
$A[j] = \emptyset$	$A[j+1] = 10$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3$	$j = 0$	$\text{key} = 10$
$A[j] = \emptyset$	$A[j+1] = 10$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3$	$j = 0$	$\text{key} = 40$
$A[j] = \emptyset$	$A[j+1] = 10$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3$	$j = 0$	$\text{key} = 40$
$A[j] = \emptyset$	$A[j+1] = 10$	



```

InsertionSort(A, n) {
    for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
        }
        A[j+1] = key
    }
}
  
```

## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3$	$j = 2$	$\text{key} = 40$
$A[j] = 30$	$A[j+1] = 40$	



```

InsertionSort(A, n) {
    for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
        }
        A[j+1] = key
    }
}
  
```

## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3$	$j = 2$	$\text{key} = 40$
$A[j] = 30$	$A[j+1] = 40$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3$	$j = 2$	$\text{key} = 40$
$A[j] = 30$	$A[j+1] = 40$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$	$j = 2$	$\text{key} = 40$
$A[j] = 30$	$A[j+1] = 40$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}
  
```

## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$	$j = 2$	$\text{key} = 20$
$A[j] = 30$	$A[j+1] = 40$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}
  
```

## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$	$j = 2$	$\text{key} = 20$
$A[j] = 30$	$A[j+1] = 40$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}
  
```



## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$	$j = 3$	$\text{key} = 20$
$A[j] = 40$	$A[j+1] = 20$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$	$j = 3$	$\text{key} = 20$
$A[j] = 40$	$A[j+1] = 20$	



```

InsertionSort(A, n) {
    for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
        }
        A[j+1] = key
    }
}
  
```

## An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4$	$j = 3$	$\text{key} = 20$
$A[j] = 40$	$A[j+1] = 40$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4$	$j = 3$	$\text{key} = 20$
$A[j] = 40$	$A[j+1] = 40$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}
  
```

## An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4$	$j = 3$	$\text{key} = 20$
$A[j] = 40$	$A[j+1] = 40$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4$	$j = 2$	$\text{key} = 20$
$A[j] = 30$	$A[j+1] = 40$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4$	$j = 2$	$\text{key} = 20$
$A[j] = 30$	$A[j+1] = 40$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

10	30	30	40
1	2	3	4

$i = 4$	$j = 2$	$\text{key} = 20$
$A[j] = 30$	$A[j+1] = 30$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	30	30	40
1	2	3	4

$i = 4$	$j = 2$	$\text{key} = 20$
$A[j] = 30$	$A[j+1] = 30$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	30	30	40
1	2	3	4

$i = 4$	$j = 1$	$key = 20$
$A[j] = 10$	$A[j+1] = 30$	



```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```

## An Example: Insertion Sort

10	30	30	40
1	2	3	4

$i = 4$	$j = 1$	$key = 20$
$A[j] = 10$	$A[j+1] = 30$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	20	30	40
1	2	3	4

$i = 4$	$j = 1$	$\text{key} = 20$
$A[j] = 10$	$A[j+1] = 20$	

```

InsertionSort(A, n) {
  for i = 2 to n {
    key = A[i]
    j = i - 1;
    while (j > 0) and (A[j] > key) {
      A[j+1] = A[j]
      j = j - 1
    }
    A[j+1] = key
  }
}

```



## An Example: Insertion Sort

10	20	30	40
1	2	3	4

$i = 4$	$j = 1$	$\text{key} = 20$
$A[j] = 10$	$A[j+1] = 20$	

```
InsertionSort(A, n) {  
  for i = 2 to n {  
    key = A[i]  
    j = i - 1;  
    while (j > 0) and (A[j] > key) {  
      A[j+1] = A[j]  
      j = j - 1  
    }  
    A[j+1] = key  
  }  
}
```

Done!

## Design of algorithms

Example:

- The divide-and-conquer approach
  - Divide the problem into a number of subproblems.
  - Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - Combine the solutions to the subproblems into the solution for the original problem.
- Recursive structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.

# Analysis of Algorithms

- Predicting the required resources
- What do we measure?
  - Computational time
  - Memory
  - Communication bandwidth
  - Other

# Analysis of Algorithms

- Analysis is performed with respect to a computational model
- We will usually use a generic uniprocessor random-access machine (RAM)
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls
  - Constant word size
    - Unless we are explicitly manipulating bits



# Input Size

- Time and space complexities
  - This is generally a function of the input size
    - e.g., sorting, multiplication
  - How we characterize input size depends:
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of vertices & edges
    - etc.

# Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call, most statements roughly require the same amount of time
    - $y = m * x + b$
    - $c = 5 / 9 * (t - 32)$
    - $z = f(x) + g(y)$

## Worst-case and average-case analysis

- worst-case running time: the longest running time for any input of size  $n$ :
  - upper bound on the running time for any input
  - for some algorithms, the worst case occurs fairly often
  - the "average case" is often roughly as bad as the worst case.
- average-case or expected running time:
  - technique of probabilistic analysis
  - assume that all inputs of a given size are equally likely
  - difficult to analyze.

## Mathematical essentials

- Monotonicity

A function  $f(n)$  is *monotonically increasing* if:

$$m \leq n \Rightarrow f(m) \leq f(n)$$

A function  $f(n)$  is *monotonically decreasing* if:

$$m \leq n \Rightarrow f(m) \geq f(n)$$

A function  $f(n)$  is *strictly increasing* if:

$$m < n \Rightarrow f(m) < f(n)$$

A function  $f(n)$  is *strictly decreasing* if

$$m < n \Rightarrow f(m) > f(n)$$

## Mathematical essentials

- Floors and ceilings

For any real number  $x$ , the greatest integer less than or equal to  $x$  is denoted by  $\lfloor x \rfloor$ .

For any real number  $x$ , the least integer greater than or equal to  $x$  is denoted by  $\lceil x \rceil$ .

For all real numbers  $x$ ,  
$$x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1.$$

Both functions are *monotonically increasing*.

# Mathematical essentials

- Exponentials

For all  $n$  and  $a \geq 1$ , the function  $a^n$  is the exponential function with base  $a$  and is *monotonically increasing*.

- Logarithms

$\log n = \log_2 n$	(binary logarithm),
$\ln n = \log_e n$	(natural logarithm),
$\log^k n = (\log n)^k$	(exponentiation),
$\log \log n = \log(\log n)$	(composition),
$\log n + k = (\log n) + k$	(precedence of log).

# Mathematical essentials

- Logarithms

For all real  $a > 0, b > 0, c > 0$ , and  $n$

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

# Mathematical essentials

- Factorials

For all  $n$  the function  $n!$  or “ $n$  factorial” is given by

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \dots \times 2 \times 1$$



# Mathematical essentials

- Functional iteration

The notation  $f^{(i)}(n)$  represents the function  $f(n)$  iteratively applied  $i$  times to an initial value of  $n$ , or, recursively

$$f^{(i)}(n) = n \text{ if } i=0$$

$$f^{(i)}(n) = f(f^{(i-1)}(n)) \text{ if } i>0$$

Example:

If  $f(n) = 2n$

then  $f^{(2)}(n) = f(2n) = 2(2n) = 2^2n$

then  $f^{(3)}(n) = f(f^{(2)}(n)) = 2(2^2n) = 2^3n$

then  $f^{(i)}(n) = 2^i n$

# Mathematical essentials

- Iterated logarithmic function

The notation  $\log^* n$  which reads “log star of  $n$ ” is defined as

$$\log^* n = \min \{ i \geq 0 : \log^{(i)} n \leq 1 \}$$

Example:

$$\log^* 2 = 1$$

$$\log^* 4 = 2$$

$$\log^* 16 = 3$$

$$\log^* 65536 = 4$$

$$\log^* 2^{65536} = 5$$

# Reading

Chapter 1 (Sections 1.1, 1.2, 1.3)

Anany Levitin, Introduction to the design and analysis of algorithms, 3rd Edition, Pearson, 2012.